

<b>Name of Student: Pushkar Sane</b>			
<b>Roll Number: 45</b>		<b>Practical Number: 9</b>	
<b>Title of Lab Assignment: To implement NumPy library in Python.</b>			
<b>DOP:</b>		<b>DOS:</b>	
<b>CO Mapped:</b> CO6	<b>PO Mapped:</b> PO3, PO5, PS01, PSO2	<b>Signature:</b>	<b>Marks:</b>

**Practical No. 9**

**Aim:** To implement NumPy library in Python.

1. Creating an array of objects using array() in NumPy.
2. Creating 2D arrays to implement Matrix Multiplication.
3. Program for Indexing and slicing in NumPy arrays.
4. To implement NumPy - Data Types.

**Description:****NumPy:**

NumPy, short for "Numerical Python," is a powerful open-source Python library used for numerical and scientific computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is a fundamental library for data manipulation in various scientific and engineering applications and is an essential building block for many other Python libraries like SciPy, pandas, and scikit-learn.

**Key Functionality of NumPy:**

1. **Multidimensional Arrays:** NumPy's core data structure is the `numpy.ndarray`, which is an efficient, homogeneous N-dimensional array that can store elements of the same data type. These arrays are highly memory-efficient and enable fast element-wise operations.
2. **Mathematical Operations:** NumPy provides a wide range of mathematical functions for array manipulation. You can perform element-wise operations, vectorized calculations, and linear algebra operations easily.

For example:

```
# Creating arrays
```

```
a = np.array([1, 2, 3])
```

```
b = np.array([4, 5, 6])
```

```
# Element-wise addition
```

```
result = a + b
```

```
print(result) # [5 7 9]
```

```
# Dot product
```

```
dot_product = np.dot(a, b)
```

```
print(dot_product) # 32
```

- 3. Broadcasting:** NumPy allows you to perform operations on arrays of different shapes. It automatically broadcasts the smaller array to match the shape of the larger array, making element-wise operations more flexible.

For example:

```
a = np.array([1, 2, 3])
```

```
b = 2
```

```
result = a + b # Broadcasting b to match the shape of a
```

```
print(result) # [3 4 5]
```

- 4. Array Slicing and Indexing:** NumPy supports powerful slicing and indexing capabilities, making it easy to extract and manipulate subsets of arrays.

For Example:

```
arr = np.array([0, 1, 2, 3, 4, 5])
```

```
# Slicing
```

```
subset = arr[2:5]
```

```
print(subset) # [2 3 4]
```

```
# Indexing
```

```
element = arr[1]
```

```
print(element) # 1
```

- 5. Random Number Generation:** NumPy includes functions for generating random numbers from various probability distributions, which is useful for simulations and statistics.

For Example

```
# Generate an array of random numbers from a uniform distribution
```

```
random_data = np.random.uniform(0, 1, size=(3, 3))
```

```
print(random_data)
```

- 6. Linear Algebra Operations:** NumPy provides functions for linear algebra operations, making it a valuable tool for tasks like matrix multiplication, eigenvalue computation, and solving systems of linear equations.

For Example:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
product = np.dot(A, B)
print(product)
```

- 7. Aggregation and Statistical Functions:** NumPy provides numerous functions for aggregating data and performing statistical calculations, including mean, median, standard deviation, variance, and more.

Example:

```
data = np.array([1, 2, 3, 4, 5])
mean = np.mean(data)
std_dev = np.std(data)
```

- 8. Array Manipulation:** NumPy allows you to reshape, transpose, concatenate, and split arrays easily. These operations are helpful for data preprocessing and manipulation.

Example:

```
array = np.array([[1, 2], [3, 4]])
# Reshape
reshaped = array.reshape((4, 1))
# Transpose
transposed = array.T
# Concatenate
concatenated = np.concatenate((array, array), axis=0)
```

NumPy's efficiency, versatility, and extensive mathematical capabilities make it an essential library for data analysis, machine learning, scientific research, and more in the Python ecosystem.

**1. Creating ndarray objects using array() in NumPy.****Code:**

```
import numpy as np

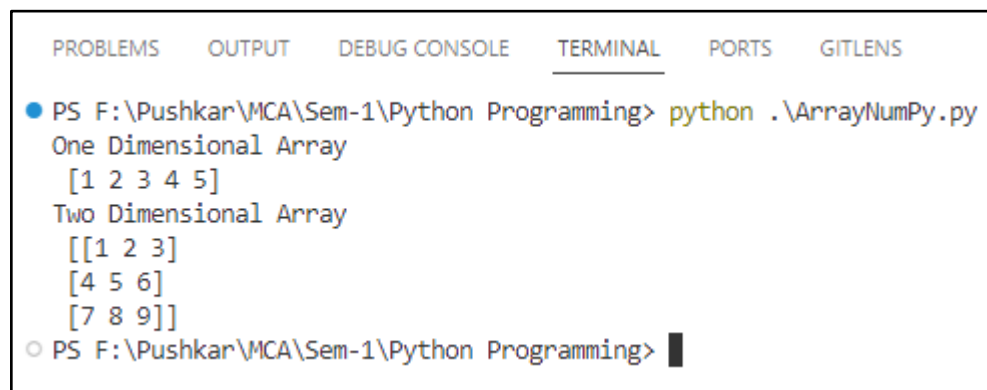
# Creating a 1D array
arr1d = np.array([1, 2, 3, 4, 5])

# Creating a 2D array (matrix)
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print ("One Dimensional Array\n", arr1d)
print ("Two Dimensional Array\n", arr2d)
```

**Conclusion:**

In this code snippet, we demonstrate the fundamental capabilities of NumPy, a powerful numerical library in Python. We start by creating both a one-dimensional array (arr1d) and a two-dimensional array or matrix (arr2d). NumPy's np.array function allows for efficient storage and manipulation of data, whether in a simple list or a more complex structure like a matrix. This code provides a basic illustration of how NumPy is used to work with arrays, which is just the tip of the iceberg when it comes to the extensive functionality that NumPy offers for numerical and scientific computing in Python.

**Output:**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ArrayNumPy.py
One Dimensional Array
[1 2 3 4 5]
Two Dimensional Array
[[1 2 3]
 [4 5 6]
 [7 8 9]]
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

## 2. Creating 2D arrays to implement Matrix Multiplication.

### Code:

```
import numpy as np

matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])

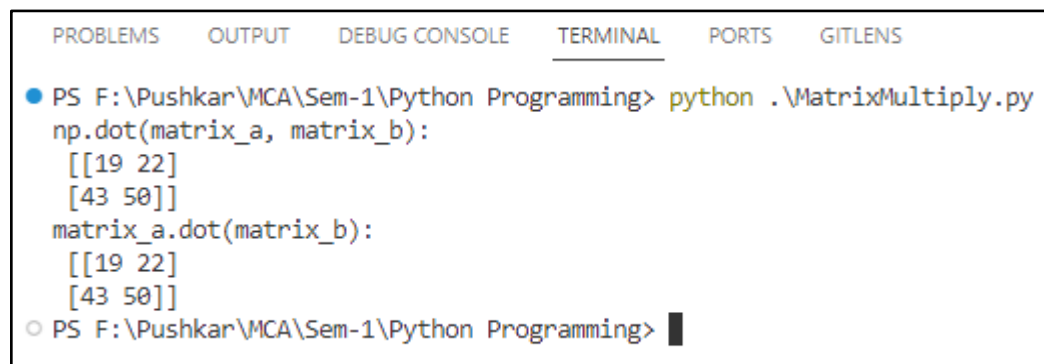
result = np.dot(matrix_a, matrix_b)
print("np.dot(matrix_a, matrix_b): \n", result)

# or
result = matrix_a.dot(matrix_b)
print("matrix_a.dot(matrix_b): \n", result)
```

### Conclusion:

In this code snippet, we delve into the power of NumPy for matrix operations. Two matrices, `matrix_a` and `matrix_b`, are created using NumPy arrays. We then utilize the `np.dot` function to perform matrix multiplication, which is a fundamental operation in linear algebra. The result is a new matrix that represents the product of the two original matrices. The code showcases the versatility of NumPy for matrix computations and provides two different ways to achieve the same result: using `np.dot(matrix_a, matrix_b)` or the dot method directly on the array, `matrix_a.dot(matrix_b)`. NumPy simplifies complex mathematical operations, making it an essential tool for scientific and engineering applications in Python.

### Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\MatrixMultiply.py
np.dot(matrix_a, matrix_b):
[[19 22]
 [43 50]]
matrix_a.dot(matrix_b):
[[19 22]
 [43 50]]
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

### 3. Program for Indexing and slicing in NumPy arrays.

**Code:**

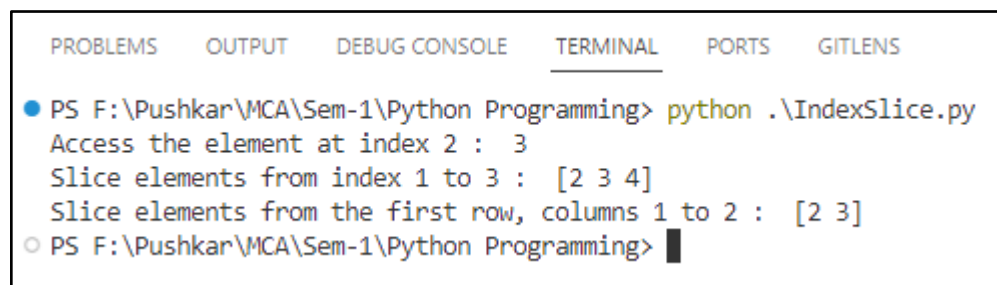
```
import numpy as np
arr1d = np.array([1, 2, 3, 4, 5])
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Indexing
element = arr1d[2] # Access the element at index 2
print("Access the element at index 2 : ", element)

# Slicing
slice1 = arr1d[1:4] # Slice elements from index 1 to 3
print("Slice elements from index 1 to 3 : ", slice1)
slice2 = arr2d[0, 1:3] # Slice elements from the first row, columns 1 to 2
print("Slice elements from the first row, columns 1 to 2 : ", slice2)
```

**Conclusion:**

In this code snippet, we explore the basics of NumPy for array indexing and slicing. Two arrays, arr1d and arr2d, are created using NumPy. The code demonstrates how to access individual elements and slices of these arrays. Using index notation, we access the element at index 2 in the one-dimensional array and slice elements from index 1 to 3. Similarly, in the two-dimensional array, we perform a more complex slice operation by specifying both row and column indices. NumPy simplifies data manipulation by providing efficient ways to access and extract specific elements or subsets from arrays, which is fundamental for data analysis and scientific computing in Python.

**Output:**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\IndexSlice.py
Access the element at index 2 : 3
Slice elements from index 1 to 3 : [2 3 4]
Slice elements from the first row, columns 1 to 2 : [2 3]
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

**4. To implement NumPy - Data Types.****Code:**

```
import numpy as np

# Specifying data types when creating arrays
arr_int = np.array([1, 2, 3], dtype=np.int32)
arr_float = np.array([1.0, 2.0, 3.0], dtype=np.float64)

# Check the data type of an array
print(arr_int.dtype)
print(arr_float.dtype)

# You can also specify the data type for individual elements
arr_custom = np.array([1, 2, 3], dtype=np.uint16)
print(arr_custom)
```

**Conclusion:**

This code snippet illustrates how to specify and work with different data types in NumPy arrays. By using the dtype parameter during array creation, you can control the data type of the elements within the array. In this example, we create arrays arr\_int and arr\_float with explicit data types of int32 and float64, respectively. We then check and print the data types of these arrays using the dtype attribute. Additionally, the code showcases that you can specify the data type for individual elements within an array, as demonstrated with the creation of the arr\_custom array with a uint16 data type. NumPy's ability to handle various data types allows for precise control over memory usage and numerical accuracy in scientific and numerical computing tasks.

**Output:**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\DataTypesNumpy.py
int32
float64
[1 2 3]
PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```